Implementation of the algebraic layer of the new APLUSIX with rewriting rules

Denis Bouhineau
Jean-François Nicaud

IRIN - Université de Nantes
2, rue de la Houssinière, BP 92208, 44322 Nantes cedex 3. FRANCE
phone : 02 51 12 58 40, fax : 02 51 12 58 12,
mail : {Denis.Bouhineau, Jean-Francois.Nicaud}@irin.univ-nantes.fr
http://www.sciences.univ-nantes.fr/info/recherche/ia/projets/aplusix

March 2000

Abstract: The APLUSIX project undertook to carry out a certain number of evolutions these last years to reach the features of a product aiming at the teaching of formal algebra, usable throughout school course for pupils from secondary school to high school. This article relates to the modification and the choices on the lowest level concerning mathematics in the system, that is the first algebraic layer of the system used to define the processing of general expressions handled in the system. At this level, a set of rewriting rules is used to describe possible mathematical handling, which is associated with a rich mechanism with parameters for matching.

General Presentation

Since 1988, the APLUSIX project aims at the modelling of human knowledge in the field of mathematics, and mode precisely in the field of formal algebra, with as a general goal the realization of interactive human learning environment used in classroom for learning mathematics. Thus, and until 1998, several prototypes have been developed on Macintosh in Lisp and regularly tested by professors of mathematics, and by psychologists; since 1998, an innovative work is done to reach the feature of a product to replace these prototypes. The results obtained with the prototypes are positive under several aspects: the pupils appreciate the system, the tests and the analysis carried out from protocols after experiments shows progress, the teachers note that the system brings a structuring of the field for the pupils; for more information about the teaching capabilities of APLUSIX see [Nguyen-Xuan and Al.-1999].

The prototype and the future product are functioning in the same way: according to two modes of interactions. The first mode is a mode of observation, where the pupil is supposed to look at the system as the system is solving a problem, the pupil can ask explanation about the action of the system. The resolution is carried out step by step, by using a pedagogical engine of resolution based on an expert system using the mathematical knowledge acquired on the level of the pupil, using simple strategic knowledge and a teacher controller to check the relevance of the action. The second mode of interaction is a mode of action, where the pupil is confronted with a problem that has to be solved step by step. The steps are to be chosen among a list of usual algebraic manipulations (use of arithmetic reduction, use of development, use of factorisation, etc.)

Explicit calculations at the arithmetic level, like calculation of the result of an addition, or a product, etc., are performed by the system itself. It guarantees that the handled

expressions are all well equivalent from an algebraic point of view, it prohibits the introduction of errors of 'calculation'. An analysis of the steps requested by the pupil is done by the system and possibly one step can be refused to the pupil if this step is erroneous from a mathematical point of view or awkward from a strategic point of view. In this mode, the pupil has the possibility to ask for help. The expert engine builds this help and take into account the resolution undertake by the pupil.

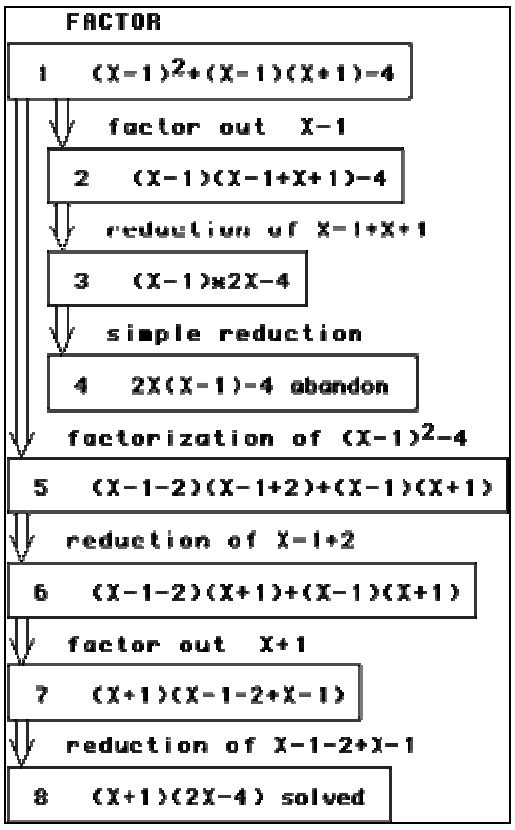

Figure 1: Layout for the prototypes of APLUSIX

In order to obtain a maximum of significant results, the prototypes of the project had some limitations. First, concerning the school level, the prototypes relate essentially to the French pupils aged from 14' to 16' (from 'quatrième' to 'seconde'). Problems concerning rational fractions, and algebraic manipulations with discriminant are not considered.

Second, with regard to the class of algebraic problems dealt with, the major class of problems concerns factorisation of polynomials and solutions of polynomial equations. Systems of equation, and inequation are not carried out.

These limitations stated, now let us note the positive points that represent particular work in the APLUSIX project. First, APLUSIX is not only a model for interactive human learning environment, or a theoretical tool for the construction and the modelling of human knowledge in situation of learning. It is also an example of software which really works and which has been used for long periods in front of pupils. In particular, with regard to the achievement in the school area, these efforts represent an important work on the interface pupil-computer and on ergonomic as for the input of expressions and for the output of theses expressions. For example, expressions in shortened, natural, form like '3x+2y=5' are accepted, but expressions or sub-expressions which are not valid from a mathematical point of view cannot be selected. For example, a pupil cannot select (with mouse or keyboard) the expression 'x+2' as a sub-

expression of the expression '3x+2y=5'. Second, for the display, algebraic expressions are drawn graphically in a two-dimensioned area. Third, at the strategic level, the development of calculations and search are done according to a tree with the possibility to backtrack, and to have several paths developed concurrently.

In term of Interactive Learning Environment, APLUSIX is halfway between the tutors and the micro-world. As a tutor, it is able to bring a pedagogical diagnosis with respect to a mathematical activity. It is able, also, to organize a sequence of exercises according to the progression of the competences of the pupil. That makes it possible to carry out a run with a professor. As a micro-world, it keeps the openness for new situations and unprepared exercises. The set of allowed algebraic manipulations is defined with parameters, the set of acceptable expressions in input is open, and solutions of one particular problem are not limited to one particular expression but mathematically defined.

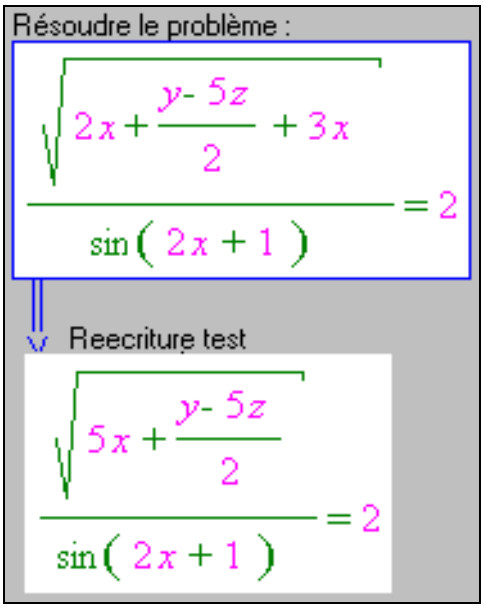

Figure 2: Layout for the new APLUSIX

Efforts have been carried out to ensure a form of transparency in the design of APLUSIX, so that the users or possible analyst can have a rather precise idea of its behaviour. A model described at knowledge-level of the prototypes was published in [Nicaud-1994]. This model is available on Internet http://www.sciences.univ-nantes.fr/info/recherche/ia/projects/aplusix. In the future product, the set of rewriting rules used to describe the behaviour of the software is accessible to the user and is written in readable form. A modification of this set –at the risk of the user- transforms the behaviour of the software. The syntax of the mathematical expression in this set is flexible and natural, as it has been expressed earlier. Attention, however, this transparency aims at an informed public, and modification of the set of rule aims at an expert public.

We hope, so, that different versions of APLUSIX adapted to particular situations can be achieved in an independent way, for example for physicists or for precise corpus of mathematics in college.

The objective for the APLUSIX project is the production of an opened system largely diffused and able to bring help in the learning in the field of formal algebra during several years, this help consisting in mainly framing the pupils to solve open exercises.

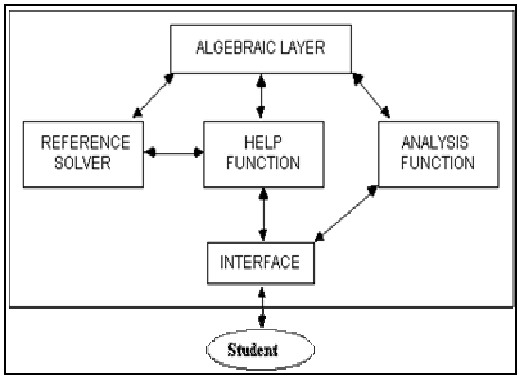

Figure 3: Architecture of APLUSIX

Rewriting rules

In the architecture of APLUSIX, see figure 3, the first layer concerning mathematics includes the definition of the structures of the data to represent the current algebraic expression and the definition of possible algebraic manipulation. Expression are represented as trees, with, for the leaves, the symbols and numerical constants, and for the nodes, the operators: functions, minus and parenthesis are considered as unary operators, fractions and equalities are considered as binary operators and sum and product are considered as n-ary operators. The algebraic manipulations are human readably described in the form of rewriting rules in a file accessible from the user among configuration files of APLUSIX.

One will note two ruptures for the APLUSIX project, on the one hand the use of rewriting rules instead of internal function based on a complete programming language (LISP), on the other hand the transparency and the accessibility of these definition at a level of knowledge less specialized.

```
RewritingRule ::=
                    Rewriting FacultativeCondition ';'
              ;

Rewriting::=
                    Expression '--->>>' Expression
              ;

FacultativeCondition::=
                    '|' Condition
              |                    // nothing
              ;

Condition ::=
                    'Type' Identifier '(' Expression ')'
              |     'Calculus' Identifier '(' Expression ')'
              |     'Calculus' Constant '(' Expression ')'
              ;
```

Figure 4: grammar for rewriting rules

The rewriting rules are described by grammar given on figure 4.

These rules comprise three parts; the last part -Condition- is optional. The first two parts are sometimes called ‘left part’ and ‘right part’. During the application of a rewriting rule, the left part can be called 'model'. If it matches with a particular expression that we will call ‘Candidate’, three results are obtained. The first result is that the rule applies to the candidate. The second result is a list of substitutions corresponding to the matching between the model and the candidate. The third result is the right part of the rewriting rule where the variables have been changed to respect the matching between the model and the candidate and the optional conditions. This second result, that is the list of substitutions, is not strictly equivalent to the third result because of the application of optional conditions.

Examples of rewriting rules are given figure 5.

```
a+0      --->>>      a;

--a      --->>>      a;

a*b+a*c  --->>>      a(b+c);

(a-b)(a+b) --->>>    a^2-b^2;

a+b --->>>           c
                | Type a (1)
                | Type b (1)
                | Calculus c (a+b);

(-a)^n --->>> -a^n
                | Type n (1)
                | Calculus 1 (Modulo n 2);

(a)+b --->>>  a+b
                | Type a (1+1);

a*b+a*c --->>>       a*d
                | Type b (1)
                | Type c (1)
                | Calculus d (b+c);
```

Figure 5: Example of rewriting rules

When a rewriting rule comprises a ‘Condition’, it is applicable only if this condition is verified. The conditions appearing in this grammar are of two different kinds, the conditions of 'type', and the conditions of ' calculus'. The conditions of 'type' require that a variable is of an algebraic particular type, like a sum, a square root, etc. This type is given ' by the example' as being the type of the expression between parentheses. It is the internal structure (that is the internal data structure) of the expression that determines it. As a tree whose nodes are typified represents each expression, the type of the root gives the type of the expression. Among the types, one finds the current types: sum, product, fraction, raise to a power, square root, constant, symbol, parenthesis, function (sine, cosine...). Minus being a unary operator, there is no type ‘difference’, but a type ‘opposite'.

The conditions of 'calculus' impose that the expression given between parentheses is calculable and that the result of this calculation can be equal to a certain expression (identifier

or constant). These conditions of 'calculus' make it possible to give values to new variables and to introduce them into the result of a rewriting (or right part of the rewriting rule) or to check numerical constraints.

Examples of application of rewriting rules are given figure 6.

Candidate	Applied rewriting rule	Resultat
3x+5+10x+3	a+b --->>>c | type a (1) | type b (1) | calculus c (a+b)	3x+8+10x
3x+8+10x	b*a+c*a --->>>a*(b+c)	x*(3+10)+8
x*(3+10)+8	a+b --->>>c | type a (1) | type b (1) | calculus c (a+b)	x*(13)+8
x*(13)+8	a*(b) --->>> b.a | type a (z) | type b(1)	13x+8

Figure 6: Application of rewriting rules

For the first example of figure 6, candidate is `3x+5+10x+3;` model is `a+b` with three conditions. The first two conditions are 'type' conditions: `a` and `b` must be numerical constants. The third condition is a 'calculus' condition `c` is the sum of `a` and `b`. The rule applies with substitution (`a:5,b:3`) that respects the 'type' condition and gives the result `3x+8+10x` after the 'calculus' condition that introduces the value `8` for the variable `c`.

The rewriting rules are grouped in classes, one rewriting rule can possibly appear in several classes. Among the classes, there is: the reductions (with the following subclasses: immediate reductions, usual reductions), factorisations (with the following subclasses: immediate factorisations, usual factorisations, factorisations according to a remarkable identity), developments, productions of squares, normal forms (for example in order to sort the monomials in a polynomial form). There will be, more or less, from ten to twenty classes or subclasses and about one hundred rewriting rules.

Examples of classes are given figure 7.

For each class of rewriting rules, several formal questions arise, as in any rewriting systems; see [Dershowitz & Jouannaud-1989] for a complete explanation if necessary. Is the rewriting system made of the rule of one or multiple classes confluent? That is, does the results of the application of different rules leads to completely different solutions? Does it terminate? That is, does the recursive application of all the possible rules finish after a finite number of steps without any rule to applies or can it be performed indefinitely? Is the system correct? That is, are the solutions given by the application of the system correct? Is the system complete? That is, can we obtain with the system all the existing solutions?

A more difficult set of questions arises when the theory of rewriting systems is confronted with educational issues. Is the rewriting systems made of the rules of one or multiple classes confluent, complete, correct and does it terminate in the context where are the pupils?

Classes	Rewriting rules

Immediate reductions	`a+0 --->>> a ;`
	`1*a --->>> a ;`
	`0*a --->>> 0 ;`
	`a^0 --->>> 1 | Type a (1)`
	`|Calculus 1 (Different a 0) ;`
	`0^n --->>> 0 | Type n (1)`
	`| Calculus 1 (Different n 0) ;`
	`a^1 --->>> a ;`
	`1^n --->>> 1 ;`
	`(-a)b --->>> -a*b ;`
	`(a*b)^n --->>> a^n*b^n ;`

Usual factorisations	`a*b+a*c --->>> a(b+c) ;`
	`b*a+c*a --->>> (b+c)a ;`
	`2a*b+a^2+b^2 --->>> (a+b)^2 ;`
	`a^2-2b*a+b^2 --->>> (a-b)^2 ;`
	`a^2-b^2 --->>> (a-b)(a+b) ;`

Productions of square	`2a*b+a^2+b^2 --->>> (a+b)^2 ;`
	`a^2-2b*a+b^2 --->>> (a-b)^2 ;`
	`a*a --->>> a^2 ;`
	`a^2 b^2 --->>> (a*b)^2 ;`
	`a^m --->>> (a^p)^2 | Type m (1)`
	`| Calculus 0 (Modulo m 2)`
	`| Calculus p (Quotient m 2) ;`
	`a --->>> c^2 | Type a (999)`
	`| Calculus 0 (QuadraticResidue a)`
	`| Calculus c (IntegerSquareRoot a) ;`

Figure 7: Classes of rewriting rules

For example, for the class of the reductions, it seems natural to ask that the rewriting system made of these rules terminates. However, this system does not have to be confluent (the expression `0+3x+5` can be reduced to `3x+5` if we reduce `0+3x` as well as it can be reduced to `5+3x` if we reduce `0+5`) Proving that this system terminates is not straightforward. And, in fact, we do not have yet made any proof. Let us explain why. If one considers only the class of the immediate reductions, already a problem consists in proving formally that these rules form a system with termination. Usually, the proof of termination for reduction systems are simple, they take their argument on the fact that the length of the character string representing the expression decreases strictly after application of any rewriting rule. Here, the class suggested on figure 7 without the rule '`(a*b)^n` → `a^n*b^n`' answers this criteria, therefore it finishes. But when we add this rule, this simple proof is not valid any more. Does this mean that the rule '`(a*b)^n` → `a^n*b^n`' is not a reduction?

Let us consider this rule '`(a*b)^n` → `a^n*b^n`': the linear form (linear opposed to the two-dimensioned form in figure 8) used for the expression of this rule answers to our

preoccupation for readability and portability, but this linear form produces an important conceptual shift in comparison to the more natural way this reduction is expressed in two-dimensioned form in figure 8. Note that in the linear representation of the rule '`(a*b)^n` → `a^n*b^n`', the number of characters on both sides of arrow is the same (and that could lead us to the conclusion that it is not a reduction), but do not correspond to the number of characters in the two-dimensioned form of figure 8. If we consider the two-dimensioned point of view the number of characters decreases, it is a reduction and so, the reduction system terminates.

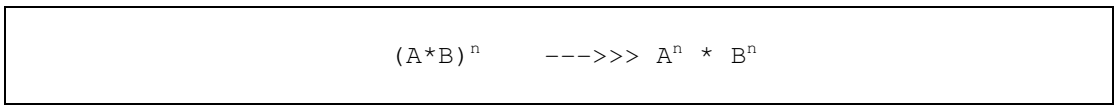

Figure 8: two-dimensioned form of the rule '`(a*b)^n` → `a^n*b^n`'

But in this two-dimensioned representation, it would have been more rigorous to say that the exponentiation introduces two pieces of information: the rise (compared to the base line) and the size reduction of the writing of the exponent. These two pieces of information are necessary to differentiate two possible interpretations for `a^b^c`: `(a^b)^c` and `a^(b^c)` as they appear on figure 9 and 9'. If one takes into account the information described here, the rewriting rule '`(a*b)^n` → `a^n*b^n`' increases the number, and then it cannot be a reduction!

Yet, for various reasons one wants that '`(a*b)^n` → `a^n*b^n`' is a reduction. Applied to polynomial: X^nY^n is a product of monomials whereas $(XY)^n$ is a complex expression with a product of monomials and a rise to the power n. Applied to 'a' a constant, for example 3 and b a monomial X: 3^nX^n can be reduce to obtain $9X^n$ whereas $(3X)^n$ cannot be reduced.

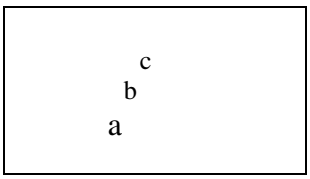

Figure 9: `(a^b)^c`

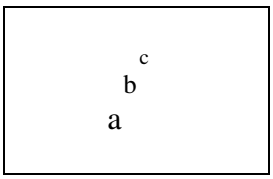

Figure 9': `a^(b^c)`

This explains the difficulty to prove formal properties on real systems used for teaching. Reality is quite complex. Other examples of formal property we would like to prove concerns classes of rule for the setting in normal form or the setting in canonical form this class must be confluent. The difficulty is the same. For the moment, we do not have given any formal proof for these properties, even if we are sure that our system owns these properties. One of our reasons is the difficulty of the proof; the second is that we do not want a syntactic (and quite technical) proof, but a semantic (and meaningful) proof because a syntactic proof would only be shared with specialists of rewriting systems and not with specialists of education, as could be a semantic proof.

In direct relationship with the rewriting rules presented here, the following section will relate to the mechanism of matching used to determine if a rule can be applied.

Mechanism for matching

The application of the rewriting rules requires the use of a mechanism of matching which, being given an expression candidate C, determines if a rule R is applicable and gives in this case a list of substitutions S for the variables of the left part of the rule (the model) corresponding to the matching between the candidate and the model.

The mechanism of matching used in APLUSIX is a form adapted to mathematics and formal algebra of the algorithms for unification which one finds in the logical or functional programming language. One can find another kind of matching algorithm in the Unix shell, and script language with joker and regular expressions that are adapted to file systems and to the processing of set of strings as in natural language data processing. See [Boizumault-1988] for unification in PROLOG and [Cousineau & Mauny-1995] for CAML. Precisely, the required matching is of the type discussed in [Borning & Bundy-1999]. Examples of the required matching are given figure 10.

Candidate	Model	Substitution
`3x+5+10x+15`	`a.c+b.c`	`a:3,b:10,c:x`
`16-(x-5)(x-5)`	a^2-b^2	`a:4,b:(x-5)`
`1+2+3+4`	`a+b`	`a:1, b:2+3+4`

Figure 10: Examples of matching

The required matching has to take into account the mathematical properties of the operators, (associative, commutative, existence of neutral elements). Matching must be able to be carried out on clean mathematical sub-expression of the candidate (that is quite classic in rewriting systems). The matching has to find substitution even if the candidate corresponds to the model multiplied by a numerical positive constant or differs only on the sign of the expression (in the following, we call this: matching with a factor). And, last, there are rewriting rules that rely on concepts or that introduce concepts, like the concept of square in the following rewriting rule 'a^2-b^2 → (a-b)(a+b)', in that case the matching must be carried out with respect to these concepts. See examples figure 10.

The stake of a rich matching is to be able to write fewer rewriting rules, more concise ones, or more general, and more natural, closer to human matching and human rewriting rules. Let's take one example, with the natural rule of factorisation expressed with '`a*b+a*c → a(b+c)`'. With a rich matching, the rewriting rule '`a*b+a*c → a(b+c)`' corresponds to the natural rule and is sufficient. With a 'poor' matching, the rewriting rule '`a*b+a*c → a(b+c)`' is not sufficient, and one must add the following set of rewriting rule; if he wants the same behaviour a human would have with '`a*b+a*c → a(b+c)`':

- To manage that the multiplication is commutative
  - `b*a+a*c → a(b+c)`
  - `a*b+c*a → a(b+c)`
  - `b*a+c*a → a(b+c)`

- To manage the introduction of neutral elements for the multiplication
  - `a+a*b` → `a(1+b)`
  - `a*b+a` → `a(b+1)`

- To manage that the multiplication is associative
  - `a*b*c+a*b*d` → `a*b(c+d)`
  - `a*b*c*d+a*b*c*d*e` → `a*b*c(d+e)`

- The fact that the addition is commutative, associative and the existence of a neutral element for addition do not require the introduction of additional rules for this example.

The risks of a too rich matching are multiple. At the data-processing level, they are expressed in terms of algorithmic complexity. As regards to educational issues, if the matching is a process independent of the mechanism of application of the rewriting rules, it is necessary to store all the information that the mechanisms of application of the rewriting rules would have allowed with respect to the didactic variables, if not there is a loss. Generally, these two mechanisms (matching and application of rewriting rule) are, more or less, equivalent from the point of view of algebraic manipulation, but this equivalence is not prolonged in the role allotted to each one of these processes in the information processing system. The rewriting mechanisms occupy a central place in APLUSIX, but the (rich) matching is 'only' a facility of expression intended to allow a certain abstraction and to follow the human reasoning more closely, and the integration of didactic information is not the same as in the application of the rewriting rules.

Consequently, there is a balance to find between richness of the matching and the size of the set of rewriting rule which one wants to handle. The more the matching is rich and the more the set of the rewriting rules will be reduced. On the other hand, the richer matching is, and the less there are catches on the didactic use of manipulations that are done, unless the matching itself relies on rewriting rules.

The technical implementation of each characteristic of the mathematical matching described previously represents considerable technical difficulties and strongly depends on the internal representations used for the expressions. The implementation also strongly depends on whether one seeks a matching starting from the candidate or from the model (it seems more efficient to carry out a common search). In all the cases, it seems necessary, on the one hand to be able to limit by program the execution of a too rich matching, on the other hand to know the mechanisms performed to get at a given matching (search for concept, use of associative, introduction of neutral elements, etc.)

Certain temporary choices were made to set up a matching of this kind in APLUSIX, with all the power we can get of it. For the removal of neutral elements, it is performed by a set of immediate reduction rewriting rules; for the moment, the commutability is carried out by duplicated rewriting rules; the associative aspect and the search for under-expressions are taken into account in the hard core of the algorithm of matching; the search for concepts (as the concept of square) is carried out in back chaining. There is nothing done for the introduction of neutral additive elements (i.e.0); the introduction of multiplicative neutral elements (i.e.1) is taken into account by the search for matching with a factor.

As for the search of matching with a factor, it is by far the most delicate mechanism to establish. It is performed with a two-pass algorithm. The first pass decorates the internal tree of the expression with possible factors appearing in the sub-expressions. The second pass tries a matching with one factor chosen among the possible ones. To limit the algorithmic

complexity, the search for matching with a factor is done only for positive factors. Matching an expression with negative factor is not taken into account for the moment, but a model for matching with a factor (positive or negative) on a different internal structure has been studied and seems to work on both positive and negative factors. In fact, the matching is the same in the new structure, but we have to change the structure we use to reach it. Perhaps it will be done in the future.

Ending's Words

APLUSIX project will obtain a final product soon. We wait impatiently to gather all the pieces to carry out the first tests in real world. In particular, it will permit to observe how our wishes to leave the most open our environment makes it possible to widen the use of it. It will permit also to tune precisely the classes of rewriting rules and the matching algorithm. And at last, it will permit to see if the new APLUSIX works as well as the old APLUSIX.

References

[Boizumault-1988] Boizumault P., PROLOG l'implantation, Masson Ed, p13-27.

[Borning & Bundy-1999] Borning A., Bundy A., Using matching in algebraic equation solving. Proceedings of IJCAI'81.

[Cousineau & Mauny-1995] Cousineau G., Mauny M., Approche Fonctionnelle de la Programmation, Ediscience (Collection Informatique), Paris.

[Dershowitz & Jouannaud-1989] Dershowitz N., Jouannaud J.P., Rewrite Systems. In handbook of theoretical computer science, vol B Chap 15, North-Holland Ed.

[Nicaud-1994] Nicaud J.F., Modélisation en EIAO, les modèles d'APLUSIX. Revue Recherche en Didactique des Mathématiques, Vol 14, n° 1-2, p 67-112.

[Nicaud et al.-1999] Nicaud, J.F., Bouhineau, D., Varlet, C., Nguyen-Xuan, A., Towards a Product for Teaching Formal Algebra. Proceedings of Artificial Intelligence and Education, Le Mans.

[Nguyen-Xuan et al.-1999] Nguyen-Xuan, A., Bastide, A., Nicaud, J.F., Learning to Solve Polynomial Factorization Problems : by Solving Problems and by Studying Examples of Problem Solving, with an ILE.. Proceedings of Artificial Intelligence and Education, Le Mans.